Fortranスマートプログラミング　― 第１回　基本的なプログラムの書き方 ―

Fortran スマートプログラミング

— 第１回　基本的なプログラムの書き方 —

田口　俊弘
摂南大学理工学部電気電子工学科

今回から３回にわたって，数値計算やコンピュータシミュレーションを目的としたFortranによる計算機プログラム作成法を説明します．最近の大学における計算機の講義ではC言語で教えるところが増えてきてFortranは影が薄くなっていますが，コンピュータシミュレーションの分野ではFortranの方が良く使われています．これは，Fortranが科学技術計算用言語として発展してきたので，数値計算に便利な書式が多数採用されているからです．例えば，複素数計算や行列計算などは文法に組み込まれているので簡単に書くことができます．

Fortranは，改版を重ねて，数値計算用としてかなり進んだ言語になっているのですが，歴史の長い言語でもあるため，古いバージョンとの互換性を保つために若干緩い文法体系になっています．例えば，デフォルトでは変数宣言をしなくてもかまわないため，変数名を書き間違えたときに発見するのが難しく，思わぬエラーを引き起こす可能性があります．これからプログラミングを始めるなら，このあたりを考慮した書き方を覚えるべきです．

そもそもコンピュータは計算する機械にすぎません．その動作は全て我々プログラマーが責任をもって指定しなければならないのです．逆に言えば，コンピュータの内部構造を理解してプログラムを書けば，効率がアップして，より高速に計算させることができます．

本稿では，コンピュータの内部構造の話を交えながら，効率が良くてエラーを見つけやすい，“スマートな”Fortran プログラムの書き方を説明します．必ずしも本稿の説明通りに書く必要はない部分もありますが,そういった部分はプログラムが長くなるにつれて御利益が現れるので,横着せずに書くことをお勧めします．

1.　パソコンでの Fortran 利用

Fortran は，計算機の歴史でいえば，その初期に開発された由緒ある言語で，“FORmula TRANslation”が語源です．当初から数値計算用言語として開発されており，筆者が計算機を利用し始めた30年前には，数値計算をするといえば，ほとんどがFortranだったのです．しかし，逆にこのことがFortran を近づきにくいものにしたようです．なぜなら，数値計算に特化するということはユーザーの専門分野が限られるということであり，その結果コンパイラの値段があまり下がらず,気軽に使うことができなかったからです.パソコンの出現とともに普及したC言語は，誕生時点から比較的安価に供給され，これが広く使われている理由の一つです．C 言語はそもそもOSを記述するために開発された言語なので計算機のハードウェア寄りの書式が多く,必ずしも初心者向きの言語だとは思えないのですが，プログラミングの専門家からすればFortranもC言語も大して変わらないので，安い方にシフトしたのはある意味やむを得ないことだと思います．

Fortranは1991年に策定されたFortran90を境に大きく変貌しました[1]．配列演算などの新しい機能を導入すると同時に，開発当初の計算機環境による制約を引きずっていた書式が緩和されて現代的な書式で書けるようになりました．例えば，それまでは１行に72文字までしか書けないとか，比較記号としての“>”や“<”が使えなかったりしたのですが，Fortran90から１行に何文字書いても良くなったし，大小記号も使えるようになりました．Fortran はその後も改版を重ね，現在の最新版はFortran2008なのですが，数値計算だけなら特に新しい文法は必要ないので，本稿ではFortran90 レベルの文法で説明をします．

さて，Fortranが使いやすくなったのと時期を同じくして，無料のOSであるLinuxやFreeBSD

が普及し始め，この OS に有志が開発した無料の Fortran（g77，gfortran など）が付属するようになりました．このため，Linux などをインストールすれば，パソコンでも Fortran が無料で使えます．このフリーの Fortran は Windows や MacOS 上で動作するものも作られていて，インターネットからダウンロードして使うことができます．数値計算プログラムを書くには Fortran の方が書きやすいし，完成したプログラムをそのまま大型計算機で高速に実行させることも可能です．

ここでは，Fortran で書いたプログラムを gfortran を使ってパソコン上で実行させる方法について紹介します．まず，Fortran プログラムを作成する時は，ファイル名の最後に“.f90”という拡張子を付けます．つまり，“文字列.f90”という名前にします．作成したプログラムファイルを計算機で実行させるには，コンピュータが直接解釈できる機械語への翻訳（コンパイル）とライブラリプログラムとの結合（リンク）という二つの過程が必要ですが，これを実行するのが gfortran コマンドです．gfortran でプログラムをコンパイルする場合，もっとも単純には，

```
$ gfortran プログラムファイル名
```

と入力します（最初の“$”はコマンドプロンプトなので入力は不要です）．例えば，test1.f90 というファイル名のプログラムをコンパイルするには，

```
$ gfortran test1.f90
```

と入力します．この時，コンパイル時に文法エラーなどが見つかるとエラーメッセージを出力して終了します．

gfortran コマンドは，コンパイルが成功すると引き続いてリンクを行い，リンクにも成功すると，“a.out”という名前のファイルを出力します．これがパソコンでプログラムを動作させることができる“実行形式ファイル”です．リンクに失敗すると，やはりエラーメッセージを出力して終了しますが，この時 a.out は作成されません．

実行形式ファイルは gfortran と同じレベルのコマンドであり，プロンプトの後に，

```
$ ./a.out
```

のようにファイルを指定することでプログラムに書かれた動作内容をコンピュータで実行させることができます．

以上が gfortran を用いたもっとも単純なプログラム実行までの流れですが，上記のような単純な命令では，どんなプログラムをコンパイルしても a.out という同じ名前の実行形式ファイルになってしまいます．また，Fortran プログラムは計算速度が重要ですから，最適化したコンパイルを行ってできるだけ高速に実行する方が良いでしょう．さらに，計算精度を考えれば倍精度実数で計算をするべきなので，自動倍精度化オプションを付加することもお勧めします．

このため，gfortran でプログラムをコンパイル・リンクする時は最低限，以下の下線部のようなオプションを付けることをお勧めします[1]．

```
$ gfortran -O -fdefault-real-8 test1.f90 -o runtest
```

ここで，“-O”（大文字のオー）が最適化のオプション，“-fdefault-real-8”が自動倍精度化のオプションです．また，“-o”（小文字のオー）のオプションの次の文字列（この例では runtest）が，実行形式ファイル名の指定です．この例では，コンパイルとリンクが正常に終了すると，“runtest”という実行形式ファイルが作成されます．よって，プログラムの実行は，

```
$ ./runtest
```

となります．

---

1 サイバーサイエンスセンターのコンパイラ f95, sxf90 ではコンパイルオプションが異なります．（センターウェブサイト参照）

1. メインプログラムの開始と終了

プログラムとはコンピュータへの動作指示（命令）を記述した文の集まりです．実行形式コマンドを入力すると，コンピュータは，

実行開始　→　プログラムに記述された命令を順に実行　→　実行終了

というステップで動作します．実行を開始した時に最初に実行する部分を“メインプログラム”といいます．言わばプログラムの本体です．プログラムには，他のプログラムの中から実行開始を指示してその機能を利用する“サブルーチン”がありますが，これについては次回説明します．

Fortran では特別な手続きをせずにプログラムを書くとそれがメインプログラムと仮定されます．しかし，それではサブルーチンと区別しにくいので，program 文を用いて最初にプログラムの名前を書きます．

```
program プログラム名
```

メインプログラムの終了は end program 文で指定します．このため，メインプログラムは次のような構造になります．

```
program code_name
   .......
   .......
   .......
end program code_name
```

この例のようにプログラム名にはアンダースコア（ _ ）も使えるので，これをスペースの代わりに使えば単語をつないだ長いプログラム名を付けることもできます．

program 文と end program 文の間に Fortran の文法に従った文を並べて動作させたい計算手順を記述します．動作手順を記述する文を“実行文”といいます．しかし，プログラムに記述するのは実行文だけではありません．計算途中で必要となる変数領域を確保するための宣言文も書かねばなりません．このような計算動作に直接携わらない文を“非実行文”といいます．Fortran では，非実行文をプログラムの最初に集約して，実行文はその後に書きます．このため，動作の開始は program 文の次の文ではなく，最初の実行文になります．

完結したメインプログラムの一例を示します．

```
program test_code
  implicit none
  real x,y,z
  x = 5
  y = 100
  z = x + y*100
  print *,x,y
  print *,' z =' ,z
end program test_code
```

このプログラムにおける実行文･非実行文の区分と,動作開始から終了までの実行の流れを図１に示します．実行形式のコマンドを入力すると，最初の実行文から動作を開始し，上から順に一行ずつ実行され，end program 文に到達した段階でプログラムの動作が終了します．後で述べる do 文を使った繰り返しや，if 文による条件分岐など，動作指定によっては所定の位置にバックしたり，条件に応じて実行するかどうかの選択ができますが，それでもそれぞれの文が終了した後に次の文が実行されるという基本的動作は変わりません．

これは計算機が一回に一つの動作しかできないためです．プログラム全体を見渡して実行するのではなく，一行一行を順に実行していくので，バックするような動作指定をしない限り，下方

に書いた実行文は上方に影響を及ぼしません．プログラムはこのことを常に念頭に置いて書かなければなりません．

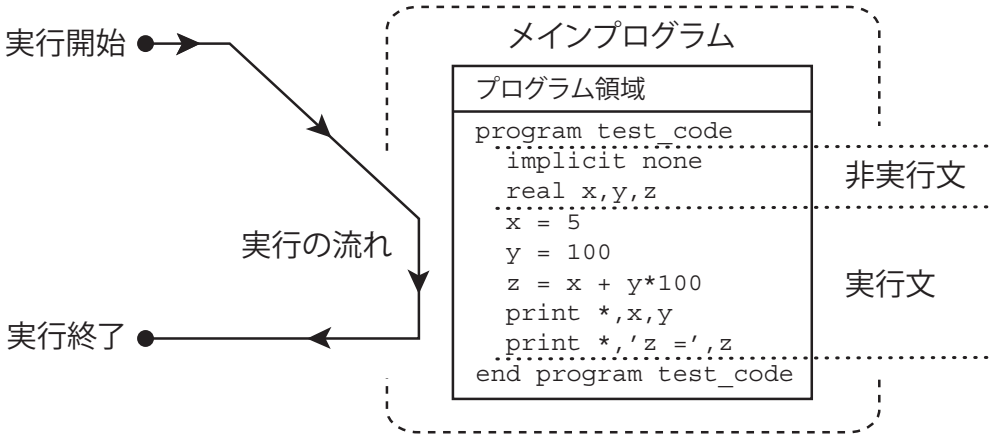

図１．プログラム開始から終了までの流れ

なお，5 節で述べる if 文を使って，条件によって途中でプログラムを終了するときや，次回お話しするサブルーチン内でプログラムを終了するときには stop 文を用います．stop 文を実行すると，その時点でプログラムが終了します．例えば，

```
program fluid_code
  implicit none
  real x,y
  integer m,n
   .......
  if (m<0) stop
   .......
end program fluid_code
```

と書けば，m＜0 の時にプログラムは終了し，それ以降は実行しません．

2. 基本的なプログラム文

2.1　代入文と演算の書式

実行文には，stop 文のように常に同じ動作を指示する文と，いくつかの“数値”を伴って，その数値に応じた動作を指示する文とがあります．プログラム中で“数値”を与える方法は３種類あります．-500 とか 3.14 のような数字を直接書く“定数”，x とか yrz のような単語で示した“変数”，およびそれらを使って x+3 や sin(y-5)のような計算手順を表した“計算式”です．計算式に書かれている手順も計算機の動作なのですが，プログラムにおいての計算式は，その計算結果を動作命令に与える“数値”として位置づけられています．このため，計算式を書いただけでは実行文になりません．

プログラムで最も良く使う実行文は“代入文”です．これは，

```
変数 = 計算式
```

という形をしています．計算式の代わりに，定数や変数を書くこともできます．プログラムにおける“変数”とはコンピュータのメモリ領域のことで，数値を入れる箱だと考えればいいでしょう．代入文は右辺の“数値”を左辺で示した変数メモリに格納する動作を表します．よって，

```
計算式 = 計算式　　！これはエラー
```

という形は使えません．Fortran において“＝”という記号は，“左辺と右辺が等しい”という意味ではないからです．例えば，

```
x = 4 + 2
y = 9 - x
y = y + 1
```

のようなプログラムを考えてみましょう．プログラムは上から順に実行されるので，まず 4+2 の結果である 6 が変数 x に代入されます．次に，9 から変数 x に保存されている数値を引いた値が変数 y に代入されるので，y には 3 が代入されます．最後の文は数学の等式と見れば変ですが，イコールが“代入する”動作だと理解できれば特に不思議な文ではありません．y はその前の文で 3 が代入されていますから，最後の文によって変数 y に y+1 の結果である 4 が代入されます．

Fortran における基本演算の書き方と使い方を表１に示します．

表１．演算記号の書き方と使い方

演算記号	演算の意味	使用例	使用例の意味
+	足し算	x+y	x＋y
-	引き算	x-y	x－y
*	掛け算	x*y	x×y
/	割り算	x/y	x÷y
**	べき乗	x**y	x^y

マイナス記号（-）は -x のように単項演算としても使えます．これらの演算には以下のような優先順位があり，優先順位の高い方が低い方より先に計算されます．

べき乗　＞　掛け算または割り算　＞　足し算または引き算

掛け算と割り算のような同レベルの演算は左から順に計算されます．さらに，かっこを使えばかっこの中が優先的に計算されます．

2.2　数値の型

コンピュータという機械が取り扱う数値は 2 進数で表されていて，基本的には整数です．例えば，1byte の数は，8bit，つまり 0 か 1 のどちらかを選択できる数が 8 個並んだものなので，10 進数では $0〜2^8-1$ (=255) までの整数を表すことができます．Fortran では小数点のない数字，56 とか -1112 などの数字を“整数型”といいます．Fortran の整数型数は 4byte（32bit）で表現されていて，$-2^{31}〜2^{31}-1$ の整数値を扱うことができます．

しかし，数値計算やシミュレーションをする場合には，3.14 のように小数点以下を含んだり，3×10^{19} とか，1.6×10^{-27} とかいった様々なスケールの数値を取り扱うため，整数型だけでは不便です．特に，整数型数は小数点以下の表現ができないので，割り算をすると全て切り捨てになります．これを忘れてプログラムを書くと，よく間違いを起こします．例えば，

```
x = (5/2)**2
y = x**(3/2)
t = 1/2*y*x
```

と書くと，x，y，z はいくつになると思いますか？答えは x=4，y=4，t=0 です．これは整数の割り算が切り捨てになるため，5/2→2，3/2→1，1/2→0 になるからです．

そこで，整数型の他に，3.14 のような少数点以下を含んだり，1.6×10^{-27} のような，$A\times10^B$ という形式の数値を取り扱うことができます．これを“実数型”といい，A の部分を“仮数部”，B の部分を“指数部”といいます．通常，数値計算は実数型で計算しなければなりません．

実数型には有効数字の違う 2 種類が用意されています．単精度実数と倍精度実数です．単精度実数とは 4byte で表される実数のことで，仮数部の有効数字は 7 桁程度です．これに対し，倍精度実数とは 8byte（64bit）で表される実数のことで，仮数部の有効数字は 15 桁程度です．7 桁あれば問題ないと思われるかもしれませんが，何百万個もの数字の合計を計算したり，次数の高い

多項式の計算をするときなどでは，計算回数の増加につれて有効桁数が減少することを考慮しなければなりません．このため，通常は倍精度実数で計算をします．

Fortran における基本的な実数型は単精度であり，倍精度実数型を使用する時にはそれを意識した書式で書かなければならないのですが，gfortran の説明で示したように最近のコンパイラはオプションを指定すればデフォルトの実数を倍精度にする“自動倍精度化機能”を持っているので，本稿では単精度実数と倍精度実数を使い分ける書式の説明は省略し，単に“実数型”と表現します．よって，特に単精度で計算する必要がなければ，コンパイル時に自動倍精度化オプションを付加して倍精度で計算して下さい．多くの計算機は倍精度実数計算を高速で行えるハードウェアを内蔵しているので，倍精度計算をしてもさほど実行速度は下がりません．

プログラム上で，整数型の定数と実数型の定数は小数点の有無で区別します．例えば，100 とか，-12345 と書けば整数型ですが，100.0 とか，-12345. とか，-0.0314 とか書けば実数型になります．また，1.6×10^{23} を入力したい時には，1.6e23 と書きます．すなわち，$A\times10^{B}$ は AeB と書きます．B が負の場合でも 1.6e-19 のように e の後に続けて書きます．また，6e20 のように eB を付加した場合には小数点が無くても実数型になります．例えば，

$$a = 3.141592\,r^2 + 3\,x^5 + 6.5\times10^{-5}\,x - 10^5$$

という式をプログラムで書けば以下のようになります．

```
a = 3.141592*r**2 + 3.0*x**5 + 6.5e-5*x - 1e5
```

この例のように，r**2 とか x**5 のように整数べき乗の指数(2 とか 5)には整数型を使います．

先ほど整数型を使ったら期待どおりの結果が出ない例を挙げましたが，以下のように実数型を使えば正しい結果が得られます．

```
x = (5.0/2.0)**2
y = x**(3.0/2.0)
y = 1.0/2.0*y*x
```

Fortran の便利な機能の一つは，複素数が使えることです．複素数にも単精度複素数型と倍精度複素数型がありますが，これも自動倍精度化機能を使えば同時に変更できるので，本稿では単に“複素数型”と表現します．複素数型の定数は，

```
(0.0,1.0)
(1e-5,-5.2e3)
(-3200.0,0.005)
```

のように，2 個の実数をコンマでつないで，かっこで囲みます．前半が実部，後半が虚部です．つまりこの例は，それぞれ，i，10^{-5}-$5.2\times10^{3}\ i$，-3200+0.005 i を表した複素数型定数です．

なお，ここまで読むと整数型は必要がないと思われるかもしれませんが，そうではありません．後で説明する配列要素を指定する数は整数型でなければならないし， do 文で用いるカウンタ変数やオン・オフを表すだけの変数など“順番”や“区別”を示すときは整数型を用います．また実数の整数部を取り出したいときや，剰余（あまり）を計算するときも整数型を利用します．数値計算プログラムの中でも整数型の使い道は多いのです．

計算式中に異なる数値型が混在した時は，精度の高い方の型に合わせて計算し，その型の値が結果になります．例えば，実数型と整数型の計算は整数型を実数型に変換して実数型と実数型の計算を行い，実数型の結果になります．複素数型と実数型の計算の場合にはその実数型を実部とした複素数型にして計算し，結果は複素数型になります．ただし，掛け算と割り算の計算は左から順に行うので注意が必要です．最初の整数型を使った計算例で，

```
t = 1/2*y*x
```

が 0 になるのは，最初に計算されるのが 1/2 という整数型÷整数型だからです．これを，

```
      t = y*x*1/2
```

と書けば，切り捨ては起こりません．

2.3　変数の宣言

次に，数式の計算結果を保存する変数の使い方を説明します．変数の名前は，頭文字が a～z のどれかであれば，後は a～z，0～9 をどの様な順序で並べたものでも使えます．例えば，abc とか k10xy 等です．Fortran では大文字と小文字を区別しないため，abc と ABC は同じ変数になります．変数にも型があり，計算結果に応じた型の変数を使わなければ正確に保存することができません．この変数の型を決める文を“宣言文”といいます．型を決めると同時に，変数のメモリ領域を確保します．このため，計算に用いる変数は全て宣言しなければなりません．宣言は一度しかできず，プログラムの実行時に変更することはできません．宣言文は非実行文です．

ところが Fortran には“暗黙の宣言”があり，通常は宣言しなくても文法的な間違いにならないので，タイプミスなどで思わぬエラーが発生する可能性があります．これを防ぐため，プログラムの２行目，すなわち program 文の次の行に必ず以下の implicit 文を書いておきます．

```
      implicit none
```

この文があれば，宣言せずに使用した変数があるとコンパイルエラーになり，タイプミスのチェックができます．

数値計算は基本的に実数で行いますが，実数型変数は次のように宣言します．

```
      real 変数名，変数名，...
```

これに対し，整数型の変数は，次のように宣言します．

```
      integer 変数名，変数名，...
```

宣言文は非実行文なので，全ての実行文より前に書かなければなりません．

```
      program test1
        implicit none
        real x,y,z,omega,wave,area
        integer i,k,n,imin,imax,kmax
         .......
```

real や integer 等の型宣言文は何行書いても良いし，順番も無関係です．

Fortran の暗黙宣言では，変数名の頭文字が a～h と o～z の変数は実数型で，i～n の変数は整数型でした．このため，整数型変数の頭文字を i～n にする習慣があります．全てを宣言する以上，基本的に変数名の付け方に制限はないのですが，整数型は用途が限られているので，頭文字を限定する方が良いと思います．実数型数の名前はあまり頭文字にこだわる必要はありませんが，原則として整数型をイメージする i,j,k,l,m,n の１文字変数は使わない方が無難です．

複素数型変数の宣言文は，

```
      complex 変数名，変数名，...
```

です．複素数型も用途が限定されているので名前の付け方に規則をつける方が良いでしょう．複素数型変数名は，頭文字を c か z にすることが多いようです．

プログラムにおいて，数値を変数に保存する意義は大別して二つあります．一つはプログラム全域にわたって共通してその値を利用するためであり，もう一つは狭い範囲で計算結果を一時的に保存するためです．この二つは意識して使い分けるようにします．特に，前者の大域的に利用する変数には長めで意味のある名前を付けるべきです．これは，１文字のような短い変数名を使うとプログラムが長くなるにつれてどこでその変数を使っているかを検索するのが難しくなるからです．

変数に数値を代入する時は，右辺の計算結果を左辺の変数の型に変換して代入します．このた

め，実数の計算結果を整数型の変数に代入すると小数点以下は切り捨てられます．例えば，

```
real x
integer n
x = 3.14
n = x + 6
```

の結果，n に代入される値は 9 です．

また，複素数型の計算結果を実数型の変数に代入すると，その複素数の実部が代入されます．例えば，

```
real x
complex c
c=(1.0,-2.0)
x=c**2
```

とすると，x=-3.00000 になります．逆に，複素数型の変数に実数型の数値を代入すると，実部に結果が代入され，虚部は 0 になります．例えば，

```
real x
complex c
x=5
c=x**2
```

とすると，c=(25.00000,0.00000)になります．

2.4　数学関数

数値計算上よく使う関数はあらかじめ用意されています．これらを“組み込み関数”といいます．代表的な数学関数を表２に示します．組み込み関数は型宣言をする必要がありません．また，引数の型に応じた精度で計算をする“総称名”機能があるので，引数 x が複素数型でも使えます．表２の必要条件と関数値の範囲は引数が実数型の場合です．この必要条件を満たさない実数型数を与えるとエラーになります．しかし複素数型を与えた場合には必ずしもエラーになりません．

表２．数学関数の書き方と使い方

組み込み関数	名　称	数学的表現	必要条件	関数値 f の範囲
sqrt(x)	平方根	$\sqrt{x}$	x≧0	
abs(x)	絶対値	$\lvert x \rvert$		
sin(x)	正弦関数	$\sin x$		
cos(x)	余弦関数	$\cos x$		
tan(x)	正接関数	$\tan x$		
asin(x)	逆正弦関数	$\sin^{-1} x$	-1≦x≦1	$-\pi/2 \leqq f \leqq \pi/2$
acos(x)	逆余弦関数	$\cos^{-1} x$	-1≦x≦1	$0 \leqq f \leqq \pi$
atan(x)	逆正接関数	$\tan^{-1} x$		$-\pi/2 \leqq f \leqq \pi/2$
atan2(y,x)	逆正接関数	$\tan^{-1}(y/x)$		$-\pi < f \leqq \pi$
exp(x)	指数関数	e^x		
log(x)	自然対数	$\log_e x$	x＞0	
log10(x)	常用対数	$\log_{10} x$	x＞0	

関数の引数には計算式を与えることも可能です．この時は，計算式の結果を引数とした関数値になります．例えば，

```
      c = sin(10*x+3) - 2*tan(-2*log(x))**3
```

のように書くことができます.

なお，$\sqrt{2}$ を計算したくて，sqrt(2)と書くとコンパイルエラーになります．なぜなら，“2”は整数型の定数であり，整数型数の平方根は用意されていないからです．必ず sqrt(2.0)のように実数型を書かなければなりません.

2.5　print 文による簡易出力

数値計算を目的としたプログラムでは，得られた計算結果を適宜表示しなければなりません．入出力の詳細については次回説明しますが，最低限，標準フォーマットの print 文を使った数値出力は覚えておく必要があります．print 文の一例を以下に示します.

```
      integer n
      n = 3
      print *, 4+5, n, n*2, 2*n-11
```

このように，“print *,”に続いて変数や計算式をコンマで区切って連ねると，それらの計算結果が横に並んで出力されます．上例の場合には，4+5 の結果である 9 から 2*n-11 の結果である-5 までが以下のように出力されます.

```
           9           3           6          -5
```

なお，“print *,”の“*”は，出力フォーマットが標準形式であることを意味しているのですが，とりあえずは形式的に書くものだと覚えて下さい.

複数の数値を出力するときに数字だけを出力すると，どれがどの数値か分からなくなる可能性があります．このような場合は，文字列を併用して変数の意味を同時に出力します．Fortran における“文字列”とは，２個のアポストロフィ「'」ではさんだ文字の並びのことで，print 文中で使えば，その文字の並びがそのまま出力されます．例えば，

```
      real x
      x = 3
      print *,'x = ',x,'  x**3 = ',x**3
```

というプログラムの出力は，

```
      x =     3.00000000000000         x**3 =     27.0000000000000
```

となります.

3. 配列

3.1　配列宣言

ここまで出てきた変数は型に応じた 1 個の数値を記憶することしかできませんでした.しかし，数値計算やシミュレーションでは，数万個のデータを保存してそれぞれを条件に応じて変化させていく，なんていうのが当たり前のように行われます．そこで，数学で $a_1, a_2, \ldots, a_n$ のように変数に添字を付けて区別するように，数字で区別した変数を作ることができます．これを“配列”といいます.

配列は変数の一種なので，型宣言文を使って宣言しておかねばなりません．単一変数と異なるのは，宣言時に添字の範囲を示す整数値を付加することです．例えば，次のように宣言します.

```
      real a(10),b(20,30)
      complex cint(10,10)
      integer node(100)
```

ここで，数字が 1 個の配列を 1 次元配列，2 個の配列を 2 次元配列といいます．3 次元以上の配

列を作ることもできます．配列の名前の付け方，頭文字の選択などの原則は単一変数名の付け方と同じにします．

宣言した配列の各部の名称は次の通りです．例えば，real a(10)と宣言した１次元配列について，a は配列全体を表す“配列名”，a(3)と書くとそれぞれのメモリを示す“配列要素”，かっこ内の数字(3)は“要素番号”とか“添字”です．

配列宣言で指定した数値は要素番号の最大値を表し，要素番号の使用可能範囲は１から指定した最大値までです．例えば a(10)の宣言では a(1)から a(10)までの 10 個の配列要素が使用可能になります．また，２次元以上の配列の場合には各次元ごとの最大値を指定しているので，b(20,30)の宣言では全部で 20×30＝600 個の配列要素が使用可能になります．

問題によっては，要素番号として 0 や負数を使いたい時があります．このような時には“:”を間に入れて，使用可能な要素番号の最小値と最大値を同時に指定します．例えば，

```
real ac(-3:5),bc(-20:20,0:100)
```

と宣言すると，１次元配列 ac は，ac(-3)から ac(5)までの 9 個が使用可能であり，２次元配列 bc は，bc(-20,0)から bc(20,100)までの(20×2+1)×(100+1)＝4141 個が使用可能です．

3.2　メモリ上での配列の並び

配列はコンピュータ内部において連続したメモリ領域で実現されています．例えば，real a(10)と宣言された配列は，図２左のように，a(1),a(2)...a(10)の順で並んだ実数型のメモリです．

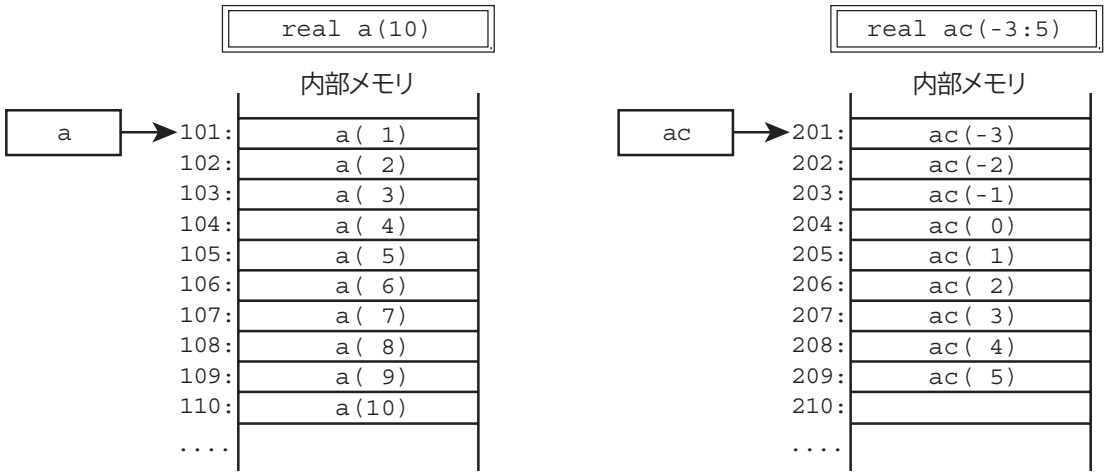

図２．１次元配列のメモリ並び

図から分かるように，a(3)とは a(1)から数えて 3 番目のメモリのことです．また，a(1000)のように範囲外の要素番号を指定すると予期せぬエラーを引き起こすこともわかります．a(10)の宣言は 10 個しかメモリを確保していないのですから，1000 番目の要素 a(1000)がどのメモリを示すのか不明だし，そもそも存在するという保証さえありません．

Fortran での配列名は配列を代表する名称であると同時に，配列の先頭メモリを示します．たとえば，配列名 a は図２左のように a(1)を示します．また，ac(-3:5)のように最小値も指定して宣言した場合には，図２右のように並んでいて，配列名 ac は ac(-3)を示します．配列名が先頭要素を示すことは配列をサブルーチンの引数として与える時に意識する必要があります．

２次元以上の配列の場合は，左の方の要素番号から先に進むようにメモリ上で並んでいます．例えば，

```
real b(3,2)
```

と宣言した場合，メモリ上での並びは図３のようになります．２次元以上の配列も配列名は先頭要素を示しています．

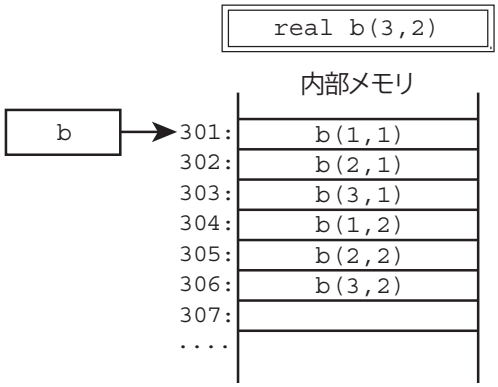

図３．２次元配列のメモリ並び

２次元と３次元の配列の並び方を式で表せば，

real b(m,n)の配列宣言で b(i,j)は先頭から数えて i+m*(j-1)番目

real b(ℓ,m,n)の配列宣言で b(i,j,k)は先頭から数えて i+ℓ*(j-1+m*(k-1))番目

となります．どちらの公式も一番右側の要素数 n に依存していません．一般的に，どんな次元の配列の公式も一番右側の要素数には依存しないのです．

4．手順の繰り返し ― do 文

実行文は基本的に上から下へ順に実行されますが，それだけでは類似した手順を繰り返す時に，必要な回数だけ同じ文を書かねばなりません．そこで，ある範囲の手順を必要な回数だけ繰り返し行わせる手段として do 文があります．do 文を使う時の基本形は，

```
do 整数型変数 = 初期値，終了値
  .......
  .......
enddo
```

です．最初の do の行が do 文で，do 文と最後の enddo 文で範囲を指定された一連の実行文が繰り返し実行されます．この範囲を“do ブロック”といいます．また，プログラムの流れが循環するという意味で“do ループ”ともいいます．do 文の整数型変数を“カウンタ変数”と呼びます．do ブロックの繰り返しは，カウンタ変数に“初期値”を代入することから開始し，do ブロック内の手順を一回実行するごとにカウンタ変数を 1 増加します．そして，カウンタ変数が“終了値”より大きくなった時点で繰り返しは終了し，enddo 文の次の文に実行が移ります．例えば，

```
do m = 1, 10
  a(m) = m
enddo
```

と書けば，a(1)～a(10)までの配列要素に，1～10 までの数値がそれぞれ代入されます．do 文における，“カウンタ変数＞終了値”の判定は do ブロックの開始時にも行うため，初期値が終了値より大きい場合にはブロック内部が一度も実行されずに enddo 文の次に実行が移ります．

次の do 文のように，整数値をもう一つ追加することで増加量を指定することもできます．

```
do 整数型変数 = 初期値，終了値，増分値
  .......
  .......
enddo
```

このときカウンタ変数は初期値から開始して，“増分値”ずつ増加しながら do ブロック内の手順

を繰り返し，“カウンタ変数＞終了値”の時点で終了します．

例えば，m が 10 以下の奇数の時のみ計算をしたい時は，

```
do m = 1, 10, 2
  .......
  .......
enddo
```

と書きます．この時の終了値は 10 ですが，10 は奇数ではないので計算しません．

増分値は負数を指定することもできます．負数の時にはカウンタ変数が減少していくので，“カウンタ変数＜終了値”になった時点で終了します．例えば，100 から順に下って 1 まで繰り返す時には以下のように書きます．

```
do m = 100, 1, -1
  .......
  .......
enddo
```

増分値が負数の時には，“初期値＜終了値”ならば do ブロック内部は一度も実行されません．

do 文のカウンタ変数を増加するタイミングは，enddo 文の実行時です．例えば，

```
do m = 1, 3
  a(m) = m**2
enddo
```

という文は，

```
m = 1
 [m＞3 の判定をする．満足しないので，do ブロックを実行]
a(m) = m**2
m = m + 1
 [m＞3 の判定をする．満足しないので，do ブロックを実行]
a(m) = m**2
m = m + 1
 [m＞3 の判定をする．満足しないので，do ブロックを実行]
a(m) = m**2
m = m + 1
 [m＞3 の判定をする．満足するので，do ブロックを終了]
```

という動作になります．この展開からわかるように，do ブロックを終了した時点で，カウンタ変数には終了値より大きい値が代入されています．上例では，do ブロック終了後，m＝4 です．do ブロック終了時の m の値を利用する時は，このことを考慮しなければなりません．

次のように，do ブロックの中に別の do ブロックを入れることもできます．ただし，カウンタ変数は異なるものを使わなければなりません．

```
do k = 1, 100
  a(k) = k**2
  do m = 1, 10
    b(m,k) = m*a(k)**3
    c(m,k) = b(m,k) + m*k
  enddo
  d(k) = a(k) + c(10,k)
enddo
```

よく使うので覚えておくと便利なのが，合計を計算する do 文です．例えば，n 要素の一次元配列，a(1)，a(2)，....，a(n)の中に数値データが入っている場合，これらの合計を求めるには do 文を使って以下のように書きます．

```
sum = 0
do m = 1, n
  sum = sum + a(m)
enddo
```

この文が合計を計算していることを理解するには，やはり以下のように手動展開してみると良いでしょう．

```
sum = 0
m = 1
sum = sum + a(m)
m = m + 1
sum = sum + a(m)
m = m + 1
sum = sum + a(m)
m = m + 1
   ......
```

ここで判定文は省略しました．このプログラムで重要なことは，do ブロックの前で変数 sum に 0 を代入していることです．これがないと正しい結果が得られないことがあります．このようなプログラムならではの書き方はパターンとして覚えておくと良いと思います．

5．条件分岐 — if 文

条件に応じて異なる手順を行わせることを“条件分岐”といい，if 文を使って指定します．もっとも単純に，一つの条件に応じて一つの文を実行するかしないかを決めるだけの時は単純 if 文を使います．単純 if 文は以下の形式です．

```
if（条件）実行文
```

単純 if 文はかっこ内の“条件”が満足されれば，その右の“実行文”を実行し，条件が満足されなければ何もしないで次の文を実行します．例えば，

```
a = 5
if (i < 0) a = 10
b = a**2
```

と書くと，i＜0 の場合には a=10 となり，それ以外は a=5 のままなので，それに応じて b に代入される値が異なります．

しかし単純 if 文には実行文が一つしか書けないので，実行したい文が複数あるときには使えません．また，条件に合った時の動作指定しかできないので，合わなかった時の動作を別に指定したい時には不便です．

そこで，通常は単純 if 文ではなく，ブロック if 文を使います．ブロック if 文とは，if 文の“実行文”のところを then にした文のことで，以下のようにブロック if 文と endif 文で一連の実行文の範囲を指定します．

```
if（条件）then
  .......
  .......
endif
```

この指定された範囲を if ブロックといい，ブロック if 文に書かれた条件を満足した時のみ，if ブロック内の実行文が実行されます．例えば，

```
a = 5
b = 2
if (i < 0) then
   a = 10
   b = 6
endif
c = a*b
```

と書くと， i＜0 の場合には a=10，b=6 の代入文を実行してから c=a*b を計算しますが，それ以外，すなわち i≧0 の場合には if ブロック内を実行しないので，a も b も変化せず，a=5，b=2 のままで c=a*b を計算します．

さて，この例では，i＜0 という条件を満足しないときには，あらかじめ a=5，b=2 という代入をする必要がありません．このように“条件を満足しないとき”に別の動作をさせたいときには if ブロック内に else 文を挿入します．else 文を挿入すると，ブロック if 文で指定した条件を満足しない場合に，else 文から endif 文までの実行文が実行されます．例えば上記のプログラムは，

```
if (i < 0) then
   a = 10
   b = 6
 else
   a = 5
   b = 2
endif
c = a*b
```

と書くことができます．この場合，i＜0 の場合には a=10，b=6 を実行し，“それ以外の場合”には a=5，b=2 を実行します．

また，条件を満足しない場合に，さらに別の条件を指定したいときには else if 文を使います．else if 文も条件はかっこで指定し，その後に then を書きます．

```
if (i < 0) then
   a = 10
   b = 6
else if (i < 5) then
   a = 4
   b = 7
else
   a = 5
   b = 2
endif
c = a*b
```

この場合，i＜0 の場合には a=10，b=6 を実行，0≦i＜5 の場合には a=4，b=7 を実行し，それ以外（i≧5）の場合は a=5，b=2 を実行，となります．else if 文による新たな条件は if ブロック内にいくつ入れてもかまいません．その場合は，“その else if 文より以前の条件を全て満足しない場合に，その条件を満足すれば”という意味になります．これに対し，else 文は最後の一回しか使えません．

数値計算でよく使う代表的な比較条件の書き方を表３に示します．

表３．比較条件の書き方

比較条件記号	記号の意味	使用例
==	左辺と右辺が等しいとき	x == 10
/=	左辺と右辺が等しくないとき	x+10 /= y-5
>	左辺が右辺より大きいとき	2*x > 1000
>=	左辺が右辺以上のとき	3*x+1 >= a(10)**2
<	左辺が右辺より小さいとき	sin(x+10) < 0.5
<=	左辺が右辺以下のとき	tan(x)+5 <= log(y)

使用例のように，比較条件を指定する場合には両辺どちらにも計算式を書くことが可能です．

さらに表４の論理演算記号を使えば，これらの条件を論理的につないだ条件や，否定した条件を与えることもできます．

表４．論理式の書き方

論理演算記号	演算の意味	使用例
“条件１”.and.“条件２”	“条件１”かつ“条件２”のとき	x > 10 .and. 2*x <= 50
“条件１”.or.“条件２”	“条件１”または“条件２”のとき	x > 0 .or. y > 0
.not.“条件”	“条件”を満足しないとき	.not.(x < 0 .and. y > 0)

例えば，

```
if (i > 0 .and. i <= 5) then
   a = 10.0
 else
   a = 0.0
endif
```

と書くと，i が 0 より大きく“かつ”5 以下の時，すなわち，0＜i≦5 のときに a=10，それ以外は a=0 となります．なお，横着してこの if 文を，

```
if (0 < i <= 5) then      ！ これはエラー
```

と書くことはできないので注意しましょう．

6．無条件ジャンプ ― goto 文，exit 文，cycle 文

プログラムというのは基本的に上から順番に実行していくものです．do 文を使えば，do ブロックで指定した範囲の実行文を所定の回数繰り返すことができますが，繰り返す範囲は固定されているし，繰り返しを終了する条件はカウンタ変数の増加で決まります．

これに対し，より一般的にプログラムの流れを変えたい時，例えば途中で計算を中断してプログラムの最初からやり直す，とか，最後の文に一気に移動して終了する，とかいう時には goto 文を使います．goto 文を使えば指定した行へ強制的に移動することができます．計算機的には，これを“ジャンプする”といいます．goto 文とは以下のような，goto の後に整数値を指定した形の文です．

```
goto 整数値
```

この goto の後の整数値がどの行にジャンプするかを指定するための数値で，“文番号”と呼ばれています．文番号はジャンプ先の行に書かれた実行文の前に，スペースを 1 個以上空けて書きます．例えば，

```
      cd = 10
      goto 11
      cd = 50
      ab = 20
      ij = 1
   11  ab = 1000
```

と書きます．最後の行で ab=1000 の前の 11 が文番号です．この例では，最初の cd=10 の実行後，cd=50 から ij=1 までの文は実行されず，直ちに ab=1000 が実行されます．すなわち，cd=50，ab=20，ij=1 の文は書いていないのと等価です．このように有無をいわさずジャンプすることを“無条件ジャンプ”といいます．

goto 文でバックすることも可能です．この場合，指定した文番号の行と goto 文の間の動作を繰り返し実行します．

```
      cd = 50
   22  ab = 200
      cd = cd + ab - ef
      ef = 10
      goto 22
      ij = 1
```

もっとも，この例ではいつまでたっても goto 文の次の ij=1 は実行されません．こういうのは“無限ループ”と呼ばれ，プログラムエラーの一つです．計算結果に応じて条件分岐し，goto 文より下の行へジャンプする別の goto 文や，プログラム自体を終了させる stop 文を挿入しなければ，プログラムは永遠に終了しません．

文番号は行の指定に使うだけなので，重複さえしなければどの行にどんな数値を付けても良いのですが，なるべく下に行くほど大きくなるように数値を選びます．また，文番号を特定の実行文に付けると，変更する時に付け替えが面倒だと思う時には continue 文を挿入します．continue 文に動作は無く，文番号で位置を指定するためだけに用います．例えば，上記のプログラムは，

```
      cd = 50
   22  continue
      ab = 200
      cd = cd + ab - ef
      ef = 10
      goto 22
      ij = 1
```

と等価です．

goto 文は，多用するとプログラムの流れがわかりにくくなるし，無限ループに陥る可能性もあるので使用はできるだけ避けるべきです．基本的な繰り返しや条件に応じたジャンプは do ブロックと if ブロックでほとんどすべて書くことができます．

do ブロックを使って繰り返し計算をする時，条件によって途中で繰り返しを終了したい場合があります．この場合，原理的には goto 文を使わなければなりませんが，goto 文の使用を極力避けるという方針から exit 文が用意されています．例えば，

```
      do m = 1, n
        sum = sum + a(m)
        if (sum > 100) exit
      enddo
      sum = sum/n
```

のように書いたプログラムは，次の goto 文を使ったプログラムと同じです．

```
      do m = 1, n
        sum = sum + a(m)
        if (sum > 100) goto 10
      enddo
   10 sum = sum/n
```

すなわち，exit 文を実行すると do ブロックの外に飛び出して enddo 文の直後から実行を開始します．なお，do ブロックの中から外へのジャンプはできますが，外から中に入るジャンプは禁止されています．

ただし，上記のプログラムを変更して，以下のように n ではなく，do ブロックを終了した時点の m で平均を計算する時には注意が必要です．

```
      do m = 1, n
        sum = sum + a(m)
        if (sum > 100) exit
      enddo
      sum = sum/m
```

なぜなら，m＝n で do ブロックを実行中に，sum＞100 の条件を満足して do ブロックの外に出た時は m＝n のままですが，条件を一回も満足せずに do ブロックを終了した時には m＝n+1 になるからです．後者の場合を考慮したプログラムにするには，do ブロックを終了した時点で m＞n かどうかを判定し，必要に応じて m を修正しなければなりません．

do ブロックの途中で実行を中断し，残りの部分をスキップしてカウンタ変数を進める時には cycle 文を使います．例えば，

```
      do m = 1, n
        sum = sum + a(m)
        if (sum > 100) cycle
        sum = sum*2
      enddo
```

のように書いたプログラムは，次の goto 文を使ったものと同じです．

```
      do m = 1, n
        sum = sum + a(m)
        if (sum > 100) goto 10
        sum = sum*2
   10 enddo
```

すなわち，cycle 文を実行するのは enddo 文にジャンプするのと等価です．enddo 文にジャンプすれば，カウンタ変数 m を増加して，m＞n かどうかをチェックした後，条件を満足しなければ再び do ブロックを最初から実行することになります．ただし，cycle 文は do ブロック内部の制御なので次のように if ブロックを使って同じ動作をさせることができます．

```
      do m = 1, n
        sum = sum + a(m)
        if (sum <= 100) then
          sum = sum*2
        endif
      enddo
```

この方が，条件に応じた動作を明示している点で良いでしょう．goto 文を多用しない方がよい，という意味合いと同じで，cycle 文を使うとプログラムがわかりにくくなるのであまり使わない

方が良いと思います．

なお，do ブロックによる繰り返しの終了を，ブロック内部の条件分岐だけで行う場合にはカウンタ変数を書かない do 文を使うことができます．例えば，

```
do
  sum = sum + x**2
  if (sum > 100) exit
  x = 1.2*x + 0.5
enddo
```

と書くと，sum が 100 を超えるまで計算を続けます．カウンタ変数なしの do ブロックは無限ループになるわけです．無限ループですから，何らかの条件分岐により外に出る記述がないと永遠に止まらないので注意して下さい．

7. プログラムのチューンアップ

計算過程が複雑になったり大量のデータを処理するようになると，数式を単純にプログラムに置きかえるのではなく，計算機の特性を考慮したプログラムにすることで計算効率や精度を高めることができます．ここでは，このような“スマートな書き方”をいくつか紹介します．

7.1　演算の速度を考える

コンピュータの計算動作は a+b のような加算が基本です．a-b のような減算は b を負数に変換して加算するだけなので加算とそれほど実行時間は変わりませんが，a*b のような乗算は加算の繰り返し動作ですから，加減算よりかなり遅いです．a/b のような除算にいたっては，減算の繰り返しを条件付きで行うので，さらに時間がかかります．この演算速度の比較を書けば次のようになります．

加減算　≫　乗算　⋙　除算

このため，割り算ができるだけ少なくなるような計算手順にします．例えば，

```
x = a/b/c
```

と書くより，

```
x = a/(b*c)
```

と書く方が速くなります．最近の計算機はかなり高速に計算することができるので，数回の計算だけならこのような書き換えはあまり意味がありませんが，do ブロック内で大量に処理をする時には価値があります．

また，do ブロック内で何度も同じ割り算をするときには，逆数を掛けるように書き換えると速くなります．例えば，

```
do i = 1, 10000
  a(i) = b(i)/c
  x(i) = a(i)/10.0
enddo
```

と書くより，

```
d = 1.0/c
do i = 1, 10000
  a(i) = b(i)*d
  x(i) = a(i)*0.1
enddo
```

と書く方が速くなります．もっとも，プログラムがわかりにくくなるという欠点もあるので，割り算の箇所が少なく，繰り返し回数がさほど多くない時にはそれほど神経質に変形する必要はないでしょう．

べき乗算はもっと遅いので，2 乗や 3 乗程度のときは掛け算にする方が速くなります．例えば，

```
x = a**2 + b**3
```

と書くより，

```
x = a*a + b*b*b
```

と書く方が速くなります．ただし，指数が大きいときには意味がないし，プログラムもわかりにくくなるので 3 乗程度までで良いと思います．

べき乗の中で，1/2 乗に関しては，組み込み関数 sqrt を利用する方が速いです．例えば，

```
y = x**0.5
z = y**1.5
```

と書くより，

```
y = sqrt(x)
z = y*sqrt(y)
```

と書く方が速くなります．

7.2　多項式を計算する手法

多項式を計算するときは，掛け算の回数ができるだけ少なくなるように考えます．一般的に，一番良い計算手法は horner 法です．例えば，

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4$$

を，乗算を明示してプログラムにすると，

```
y = a0 + a1*x + a2*x*x + a3*x*x*x + a4*x*x*x*x
```

のようになり，10 回の乗算が必要です．ところが，これを変形して，

```
y = a0 + (a1 + (a2 + (a3 + a4*x)*x)*x)*x
```

と書けば，4 回の乗算で計算できます．これが horner 法です．一般的に，horner 法は 0 の係数が少ない多項式の計算に有効です．もし n 次の多項式，

$$y = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

において，係数 $a_0, a_1, a_2, \ldots, a_n$ が，a(0), a(1), a(2), . . . , a(n) という配列に入っている場合には，次のような do 文を使って計算することができます．

```
y = a(n)
do i = n-1, 0, -1
  y = a(i) + x*y
enddo
```

7.3　同じ計算は繰り返さない

do ブロック内部で同じ計算を繰り返す時には，あらかじめ計算をしておきます．例えば，

```
do i = 1, 10000
  a(i) = b(i)*c*f
  b(i) = sin(x)*a(i)
enddo
```

と書くと，繰り返しごとに c*f や sin(x)を計算することになりますが，これらは do ブロック内部では変化しないのですから，あらかじめ計算しておくと速くなります．例えば，

```
cf = c*f
sx = sin(x)
do i = 1, 10000
  a(i) = b(i)*cf
  b(i) = sx*a(i)
enddo
```

のように書き換えると速くなります．

7.4　do 文を多重にするときの順序

配列を使った繰り返し計算をするときは，メモリをできるだけ連続的に読み書きする方が速くなります．このため，２次元以上の配列計算をするときには左の要素から順に進めるようにします．例えば，

```
real b(10,100)
integer m,n
do n = 1, 100
  do m = 1, 10
    b(m,n) = m*n
  enddo
enddo
```

のように，２次元配列 b(m,n)に計算結果を代入するときは，左側の要素 m に関する do ブロックを右側の要素 n の do ブロックの内側に持ってくる方が高速です．これは，内側の do ブロックのカウンタ変数が先に進むので，b(1,1),b(2,1),b(3,1)...のように，メモリの並んでいる順に格納していくからです．これを，

```
real b(10,100)
integer m,n
do m = 1, 10
  do n = 1, 100
    b(m,n) = m*n
  enddo
enddo
```

のように書くと,b(1,1),b(1,2),b(1,3)...のように飛び飛びに格納していくので効率が悪くなり，スピードダウンします．

7.5　桁落ちに気を付ける

実数型数の有効数字は倍精度でも 15 桁程度です．よって，値の接近した 2 個の実数の引き算をするときは気を付けなければなりません．例えば，

2000.06 − 2000.00 = 0.06

ですが，計算結果 0.06 は元の 2000.06 と比べると有効数字が 5 桁も少なくなっています．これを“桁落ち”といいます．桁落ちするような引き算はできるだけ避けねばなりません．

例えば，２次方程式 $ax^2 + bx + c = 0$ の１根は，

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

ですが，$b > 0$ かつ $b^2 \gg |4ac|$ の時には，b と $\sqrt{b^2 - 4ac}$ がほぼ等しくなるので，$-b + \sqrt{b^2 - 4ac}$ の

計算をすると桁落ちする可能性があります．そこでこの根を計算する時は，分子の有理化をします．すなわち，分母分子に $-b-\sqrt{b^2-4ac}$ を掛けると，

$$x_1=\frac{\left(-b+\sqrt{b^2-4ac}\right)\left(-b-\sqrt{b^2-4ac}\right)}{2a\left(-b-\sqrt{b^2-4ac}\right)}=\frac{2c}{-b-\sqrt{b^2-4ac}}$$

となりますが，最後の公式を使えば桁落ちする心配がありません．2 根とも計算する時には，2 根の積が c/a であることを利用して，まず桁落ちしない方の根 x_1 を計算した後で，もう一方の根を $c/(ax_1)$ で計算すると良いでしょう．

大きな数に小さい数を加えると，小さい数が桁落ちして情報が失われる可能性があります．例えば，１次元配列 a(i)の合計 s を計算するプログラムは，

```
s = 0
do i = 1, n
  s = s + a(i)
enddo
```

で良いのですが，a(i)が全て正数で，n が非常に大きい場合には，合計 s がだんだん大きくなって，後の方で加えた a(i)の情報が失われる可能性があります．倍精度実数を使うことを推奨している理由の一つがこれです．この桁落ちを防ぐには，例えば，2 個ずつ加えてそれらを別の配列に入れ，その後それらを同様に 2 個ずつ加えていって，...とするのが良いのですが，プログラムが複雑になってしまいます．

ひとつの比較的簡単な解決法は，補助変数 r を使って，誤差を別に評価しておく方法です[2]．

```
s = 0
r = 0
do i = 1, n
  r = r + a(i)
  t = s
  s = s + r
  t = s - t
  r = r - t
enddo
```

この方法では，桁落ち分を変数 r に保存して別途加えるので，計算精度が上がります．倍精度計算をしていれば必ずしもこの方法にする必要はありませんが，精度の良い合計計算が必要になった時のために覚えておくと良い方法です．

桁落ちしそうな引き算は可能な限り避けるようにします．例えば，小さい正数の dt(i)という１次元配列が与えられていて，これから，

```
t(1) = 0.0
do i = 2, n
  t(i) = t(i-1) + dt(i)
enddo
```

により t(i)という１次元配列を作ったとします．この t(i)を使って t(i+1)-t(i)や t(i+1)-t(i-1)の値が必要な時には，dt(i)も保存しておいて dt(i)や dt(i)+dt(i-1)を使って評価するべきです．

8. 読みやすいプログラムを書くには

ここではプログラムを読みやすくするためのヒントをいくつか紹介します．プログラムは，１文字書き間違えただけでも結果が全く異なることがあり，書くことよりも誤りがないことをチェックする方が大変です．“読みやすいプログラム”というのは，チェックがしやすいプログラムのことです．読みやすく書くように心がけていれば，書き間違いをしても気がつきやすいし，実行時にエラーが出ても早期に間違っている箇所を発見することができます．以下のヒントはプログラミングの初心者にはその重要性がわからず，“細かいことだから無視してもいいや”と感じるかもしれません．しかし，読みやすいプログラムにすることは，プログラムが長くなるほど意義が出ますし，慣れればそれほどの手間ではありません．面倒がらずに心がけましょう．

8.1　コメント文を多用しよう

Fortran では“!”の後に続く文字列は全て無視されます．すなわち何を書いても実行とは無関係です．これをコメント文といいます．コメント文を機会あるごとにプログラム中に入れて，書かれている内容を表示するように心がけましょう．さもなくば，プログラムが長くなるにつれて，自分でも何を書いたのか忘れてしまいます．例えば，

```
! area of circles
  s = pi*r*r
```

のように，１列目に ！ を書けば，その行はコメント行となります．また，

```
  s = pi*r*r          ! 円の面積
  v = 3*pi*r*r*r/4    ! 球の体積
```

のように，実行文の末尾に書き込むこともできるし，日本語を書くこともできます．ただし，日本語環境によっては正しく認識できずに文字化けすることがあるので，可般性を考慮するならローマ字つづりでも良いから半角英数字だけでコメントを書いた方が無難です．

8.2　ブロックを明確にするために字下げする

今までの例でも示していますが，do や if などのブロック中では字下げ（インデント）をしましょう．これは，ブロックの範囲が一目でわかるからです．

例えば，if ブロック，

```
  if (n < 0) then
     x = 10.0
   else if (n < 10) then
     x = 1.0
   else
     x = 0.0
  endif
```

や do ブロック，

```
  do i = 1, n
    b(i) = a(i)
    c(i) = a(i)*sin(b(x))
  enddo
```

のように，ブロック内部の文をスペースを入れてずらせておくとブロックの範囲が一目で判断できます．通常，2〜4 個のスペース程度で良いと思います．

最近のエディタには，オートインデントといって，改行するとその前の行と先頭がそろうようにカーソルが移動する機能があるので，インデントをするのはそれほどの手間ではありません．

8.3　文字間や行間は適当に空ける

文中の文字間は適当に空けたほうが読みやすくなります．筆者は“=”と“+”の両側に必ずスペースを入れることにしています．また，代入文が何行も続くときには“=”を上下にそろえると読みやすくなります．特に，同じパターンで少しずつ内容が違う文を並べるときには，パターンが並ぶようにスペースを入れることをお勧めします．以下は，筆者のシミュレーションプログラムの１部です．

```
      wm( 1,m1) = (1-dx)*(1-dy)*(1-dz)
      wm( 2,m1) =    dx *(1-dy)*(1-dz)
      wm( 3,m1) = (1-dx)*   dy *(1-dz)
      wm( 4,m1) =    dx *   dy *(1-dz)
      wm(11,m1) = (1-dx)*(1-dy)*   dz
      wm(12,m1) =    dx *(1-dy)*   dz
      wm(13,m1) = (1-dx)*   dy *   dz
      wm(14,m1) =    dx *   dy *   dz
```

これをスペース抜きで書くと，以下のようになります．

```
      wm(1,m1)=(1-dx)*(1-dy)*(1-dz)
      wm(2,m1)=dx*(1-dy)*(1-dz)
      wm(3,m1)=(1-dx)*dy*(1-dz)
      wm(4,m1)=dx*dy*(1-dz)
      wm(11,m1)=(1-dx)*(1-dy)*dz
      wm(12,m1)=dx*(1-dy)*dz
      wm(13,m1)=(1-dx)*dy*dz
      wm(14,m1)=dx*dy*dz
```

この二つは全く同じことが書いてあるのですから全く同じ結果を出します．しかし，パターンをそろえた前者は単なる美的趣味でスペースを入れたのではありません．この例では，どこか一箇所，例えば dx を dy に書き間違えただけでも正しい結果が得られません．しかも，dx と dy を書き間違えても文法的なミスにはならず，出力結果を見ても間違いに気づかない可能性があります．よって，できるだけプログラムを書いている段階でミスを取り除いておかねばなりません．

この時，前者のようにパターンをそろえておけば，横方向に一行一行チェックするだけでなく縦方向にも比較しながらチェックすることができます．色々な角度からチェックすることで，ミスのないプログラムにすることができるのです．

この他，文と文の間に適当なコメント文や，何も書いていない行を挿入すれば，プログラムの流れに区切りができてわかりやすくなります．これは文章を段落に分けるように，プログラムを区分けすると考えればいいでしょう．

8.4　意味不明の定数はできるだけ減らそう

数学的に決まっている単純な定数（2 倍するとか，1 を加えるとか，3 乗するとか）の場合以外は，できるだけ定数の使用量を減らすほうがいいと思います．プログラムを書いている時には理解していても時間が経ってから読んだ時に意味が分からなくなる可能性があるからです．

例えば，電子の質量 m_e（=9.1093897×10^{-31}kg）と光の速度 c（=2.99792458×10^{8}m/s）を使って計算すれば m_ec^2 = 8.187111168×10^{-14}（J）ですが，これを単位とするエネルギー，$energy/m_ec^2$ を計算するプログラムを，

```
      ene = energy/8.187111168e-14
```

と書くと，後で 8.187111168e-14 という数字がどこから来たのか分からなくなる可能性があります．こういう時には多少面倒でも，

```
me  = 9.1093897e-31
cl  = 2.99792458e8
mc2 = me*cl**2
ene = energy/mc2
```

としておけば，わかりやすいしプログラム内容の記録にもなります．

また，do ブロックを 100 回繰り返す，とか，10 回ごとに出力する，とかいう制御数も，変数にしておけば数値の意味が明示されるし，変更もしやすくなります．例えば，

```
do i = 1, 100
  a(i) = b(i)**2
enddo
sum = 0
do i = 1, 100
  sum = sum + a(i)
enddo
```

という do ブロックにおいて，100 という数は配列要素数に依存する共通の数値ですから，以下のように変数にします．

```
imax = 100
do i = 1, imax
  a(i) = b(i)**2
enddo
sum = 0
do i = 1, imax
  sum = sum + a(i)
enddo
```

こうしておけば意味がはっきりするし，100 を 200 に変更したい時には変更点が一箇所ですみます．このような変数化はプログラムが長くなるほど重要性が増します．

まとめと次回の予告

今回は，基本的なプログラムの書き方と，書く時の“こつ”について説明しました．今回説明した基本的文法を使うだけでも，様々な数値計算プログラムを書くことができます．昔から“習うより慣れろ”といいます．身近なパソコンを利用して色々な計算をしてみて下さい．

次回は，サブルーチンについて説明します．サブルーチンを使えば，一連の計算手順をブラックボックス化してその機能だけを使ったり，大きなプログラムを分割してメンテナンスのしやすいプログラムにすることができます．

また，数値を出力する際の出力形式を指定する方法や，データをファイルに保存したりファイルから読み込んだりする方法など，入出力文の詳細についても説明します．

参考文献

[1] 入門 Fortran90 実践プログラミング，東田幸樹・山本芳人・熊沢友信，ソフトバンク，1994 年
[2] 数値計算の常識，伊理正夫・藤野和建，共立出版，1985 年